PATENT APPLICATION

EXTERNAL MEMORY CONTROLLER NODE

Inventor(s):   Frederick Curtis Furtek, a citizen of the United States, residing at:
2470 Sharon Oaks Drive
Menlo Park, CA 94025

Inventor(s):   Paul L. Master, a citizen of the United States, residing at:
796 Dartshire Way
Sunnyvale, CA 94087

EXTERNAL MEMORY CONTROLLER NODE

CLAIM OF PRIORITY

[01]   This application claims priority from U.S. Provisional Patent Application No. 60/428,646, filed on November 22, 2002; that is hereby incorporated by reference as if set forth in full in this application for all purposes.

CROSS REFERENCE TO RELATED APPLICATIONS

[02]   This application is related to the following U.S. patent applications, each of which is hereby incorporated by reference as if set forth in full in this document for all purposes:

Serial no. 09/815,122, entitled "Adaptive Integrated Circuitry with Heterogeneous and Reconfigurable Matrices of Diverse and Adaptive Computational Units having Fixed, Application Specific Computational Elements," filed on March 22, 2001;

Serial no. 10/443,554, entitled "Uniform Interface for a Functional Node in an Adaptive Computing Engine," filed on May 21, 2003.

BACKGROUND OF THE INVENTION

[03]   The present invention is related in general to memory controllers and more specifically to the design of a memory controller for use in an adaptive computing environment.

[04]   The advances made in the design and development of integrated circuits ("ICs") have generally produced information-processing devices falling into one of several distinct types or categories having different properties and functions, such as microprocessors and digital signal processors ("DSPs"), application specific integrated circuits ("ASICs"), and field programmable gate

arrays ("FPGAs"). Each of these different types or categories of information-processing devices have distinct advantages and disadvantages.

[05]   Microprocessors and DSPs, for example, typically provide a flexible, software-programmable solution for a wide variety of tasks. The flexibility of these devices requires a large amount of instruction decoding and processing, resulting in a comparatively small amount of processing resources devoted to actual algorithmic operations. Consequently, microprocessors and DSPs require significant processing resources, in the form of clock speed or silicon area, and consume significantly more power compared with other types of devices.

[06]   ASICs, while having comparative advantages in power consumption and size, use a fixed, "hard-wired" implementation of transistors to implement one or a small group of highly specific tasks. ASICs typically perform these tasks quite effectively; however, ASICs are not readily changeable, essentially requiring new masks and fabrication to realize any modifications to the intended tasks.

[07]   FPGAs allow a degree of post-fabrication modification, enabling some design and programming flexibility. FPGAs are comprised of small, repeating arrays of identical logic devices surrounded by several levels of programmable interconnects. Functions are implemented by configuring the interconnects to connect the logic devices in particular sequences and arrangements. Although FPGAs can be reconfigured after fabrication, the reconfiguring process is comparatively slow and is unsuitable for most real-time, immediate applications. Additionally, FPGAs are very expensive and very inefficient for implementation of particular functions. An algorithmic operation implemented on an FPGA may require orders of magnitude more silicon area, processing time, and power than its ASIC counterpart, particularly when the algorithm is a poor fit to the FPGA's array of homogeneous logic devices.

[08]   An adaptive computing engine (ACE) or adaptable computing machine (ACM) allows a collection of hardware resources to be rapidly configured for different tasks. Resources can include, e.g., processors, or nodes,

for performing arithmetic, logical and other functions. The nodes are provided with an interconnection system that allows communication among nodes and communication with resources such as memory, input/output ports, etc. One type of valuable processing is memory access services. In order to provide memory access services to access external memory, an external memory controller is typically needed.

[09]   Thus, there is a desire to provide a memory controller that provides memory access services in an adaptive computing engine.

BRIEF SUMMARY OF THE INVENTION

[10]   Embodiments of the present invention generally relate to using a memory controller to provide memory access services in an adaptive computing engine.

[11]   In one embodiment, a memory controller in an adaptive computing engine (ACE) is provided. The controller includes a network interface configured to receive a memory request from a programmable network; and a memory interface configured to access a memory to fulfill the memory request from the programmable network, wherein the memory interface receives and provides data for the memory request to the network interface, the network interface configured to send data to and receive data from the programmable network.

[12]   In another embodiment, a memory controller includes a network interface configured to receive a memory request for a memory access service from a network; and one or more engines configured to receive the memory request and to provide the memory access service associated with the memory request.

[13]   In yet another embodiment, a memory controller includes one or more ports configured to receive memory requests, wherein each port includes one or more parameters; an engine configured to receive a memory request from a port in the one or more ports; and a data address generator configured to generate a memory location for a memory based on the one or more parameters

4

associated with the port, wherein the engine is configured to perform a memory operation for the memory request using the generated memory location.

[14] In another embodiment, a memory controller includes one or more ports configured to receive memory requests from requesting nodes, wherein each port includes one or more parameters, the one or more parameters configurable by information in the memory requests; a point-to-point engine configured to receive a memory request from a port in the one or more ports; a data address generator configured to generate a memory location for a memory based on the one or more parameters associated with the port, wherein the point-to-point engine performs a memory operation using the generated memory location while adhering to a point-to-point protocol with the requesting node.

[15] In another embodiment, a system for processing memory service requests in an adaptable computing environment is provided. The system comprises: a memory; one or more nodes configured to generate a memory service request; a memory controller configured to receive the memory service request, the memory controller configured to service the memory service request, wherein the memory controller reads or writes data from or to the memory based on the memory service request.

[16] A further understanding of the nature and the advantages of the inventions disclosed herein may be realized by reference of the remaining portions of the specification and the attached drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates an embodiment of an ACE device;

Fig. 2 shows a plurality of ACE devices, each having a plurality of nodes, connected together in a development system;

Fig. 3 is a block diagram of a system for performing memory access services according to one embodiment of the present invention;

Fig. 4 illustrates a more detailed block diagram of memory controller according to one embodiment of the present invention; and

Fig. 5 illustrates an embodiment of a point-to-point (PTP) engine usable to perform PTP memory services according to the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[17]   A preferred embodiment of the invention uses an adaptive computing engine (ACE) architecture including an external memory controller (XMC) node. Details of an exemplary ACE architecture are disclosed in the U.S. patent application serial no. 09/815,122, entitled "Adaptive Integrated Circuitry with Heterogeneous and Reconfigurable Matrices of Diverse and Adaptive Computational Units having Fixed, Application Specific Computational Elements," referenced, above.

[18]   In general, the ACE architecture includes a plurality of heterogeneous computational elements coupled together via a programmable interconnection network. Figure 1 illustrates an embodiment 100 of an ACE device. In this embodiment, the ACE device is realized on a single integrated circuit. A system bus interface 102 is provided for communication with external systems via an external system bus. A network input interface 104 is provided to send and receive real-time data. An external memory interface 106 is provided to enable the use of additional external memory devices, including SDRAM or flash memory devices. A network output interface 108 is provided for optionally communicating with additional ACE devices, as discussed below with respect to Figure 2.

[19]   A plurality of heterogeneous computational elements (or nodes), including computing elements 120, 122, 124, and 126, comprise fixed and differing architectures corresponding to different algorithmic functions. Each node is specifically adapted to implement one of many different categories or types of functions, such as internal memory, logic and bit-level functions, arithmetic functions, control functions, and input and output functions. The quantity of nodes of differing types in an ACE device can vary according to the application requirements.

[20]   Because each node has a fixed architecture specifically adapted to its intended function, nodes approach the algorithmic efficiency of ASIC devices. For example, a binary logical node may be especially suited for bit-manipulation operations such as, logical AND, OR, NOR, XOR operations, bit shifting, etc. An arithmetic node may be especially well suited for math operations such as addition, subtraction, multiplication, division, etc. Other types of nodes are possible that can be designed for optimal processing of specific types.

[21]   Programmable interconnection network 110 enables communication among a plurality of nodes such as 120, 122, 124 and 126, and interfaces 102, 104, 106, and 108. The programmable interconnection network can be used to reconfigure the ACE device for a variety of different tasks. For example, changing the configuration of the interconnections between nodes can allow the same set of heterogeneous nodes to implement different functions, such as linear or non-linear algorithmic operations, finite state machine operations, memory operations, bit-level manipulations, fast-Fourier or discrete-cosine transformations, and many other high level processing functions for advanced computing, signal processing, and communications applications.

[22]   In one embodiment, programmable interconnection network 110comprises a network root 130 and a plurality of crosspoint switches, including switches 132 and 134. In one embodiment, programmable interconnection network 110 is logically and/or physically arranged as a hierarchical tree to maximize distribution efficiency. In this embodiment, a number of nodes can be clustered together around a single crosspoint switch. The crosspoint switch is further connected with additional crosspoint switches, which facilitate communication between nodes in different clusters. For example, cluster 112, which comprises nodes 120, 122, 124, and 126, is connected with crosspoint switch 132 to enable communication with the nodes of clusters 114, 116, and 118. Crosspoint switch is further connected with additional crosspoint switches, for example crosspoint switch 134 via network root 130, to enable communication between any of the plurality of nodes in ACE device 100.

[23]   The programmable interconnection network (PIN) 110, in addition to facilitating communications between nodes within ACE device 100, also enables communication with nodes within other ACE devices via network inputs and outputs interfaces 104 and 108, respectively, and with other components and resources through other interfaces such as 102 and 106. Figure 2 shows a plurality of ACE devices 202, 204, 206, and 208, each having a plurality of nodes, connected together in a development system 200. The system bus interface of ACE device 202 communicates with external systems via an external system bus. Real-time input is communicated to and from ACE device 202 via a network input interface 210. Real-time inputs and additional data generated by ACE device 202 can be further communicated to ACE device 204 via network output interface 212 and network input interface 214. ACE device 204 communicates real-time inputs and additional data generated by either itself or ACE device 202 to ACE device 206 via network output interface 216. In this manner, any number of ACE devices may be coupled together to operate in parallel. Additionally, the network output interface 218 of the last ACE device in the series, ACE device 208, communicates real-time data output and optionally forms a data feedback loop with ACE device 202 via multiplexer 220.

[24]   In accordance with embodiments of the present invention, a memory controller is used to provide memory access services in an ACE architecture. Fig. 3 is a high-level block diagram that illustrates the basic concepts of a system 300 for performing memory access services according to one embodiment of the present invention. As shown, system 300 includes PIN 110, nodes 301, a memory controller 302, and a memory 304.

[25]   Nodes 301 can be any nodes, (e.g., computational elements or resources) in a computing device. Nodes 301 initiate memory service requests to memory controller 302. For example, nodes 301 can initiate read and write commands. If a read command is initiated, the requesting node is considered a "consumer" in that it consumes data read from memory 304 and if a write command is initiated, the requesting node is considered a "producer" in that it

produces data to be written to memory 304. The read and write commands may be in the form of different memory access services that are described below.

[26]   PIN 110 receives memory service requests from nodes 301 in the ACE device. Additionally, PIN 110 receives and/or sends data from/to memory controller 302 and receives and/or sends the data from/to the requesting nodes in the ACE device.

[27]   Memory controller 302 receives memory access service requests from PIN 110 and processes the requests accordingly. In one embodiment, the services provided by memory controller 302 include a peek and poke service, a memory random access (MRA) service, a direct memory access (DMA) service, a point-to-point (PTP) service, a real-time input (RTI) service and a message service. The peek and poke service allows a requesting node to peek (retrieve) data and poke (write) data found in memory controller 302. A memory random access (MRA) service allows a requesting node to do a read and write to memory 304. A direct memory access (DMA) service allows a requesting node to request large blocks of data from memory 304. A point-to-point (PTP) service allows a requesting node to read and write data, and update port parameters, in a process that conforms to a point-to-point protocol. In one embodiment, the PTP service is used to read and write real-time streaming data. The real-time input (RTI) service performs the same service as to PTP service but uses a reduced acknowledgement protocol. Additionally, memory controller 304 provides messaging to nodes in the ACE device. For example, memory controller 302 can provide confirmation acknowledgement messages to requesting nodes that may be used for flow control.

[28]   In one embodiment, memory 304 is an external memory for an ACE device. Memory 304 receives memory service requests from memory controller 302 and provides data to memory controller 302 when a read operation is requested. Additionally, memory controller 302 may provide data to memory 304 that is to be written to memory 304. Memory 304 may be any memory, such as, a synchronous dynamic random access memory (SDRAM), a flash memory, static random access memory (SRAM) and the like.

[29]   The above-mentioned services that may be provided by memory controller 302 will now be described. Although the following memory services are described, it will be understood that a person skilled in the art will appreciate other memory services that memory controller 302 may provide.

[30]   Flow control is provided for a poke request in that a requesting poke waits for a poke acknowledgement before initiating a new poke to the same memory. In the case where multiple services are provided in memory 304, multiple requests to different memories may be allowed.

[31]   Fig. 4 illustrates a more detailed block diagram of memory controller 302 according to one embodiment of the present invention. As shown, memory controller 302 includes a PIN interface 400, one or more engines 402, and a memory interface 404. Additionally memory 304 includes an SDRAM memory 406 and a flash memory 408.

[32]   PIN interface 400 is configured to receive memory service requests from PIN 110. Additionally, PIN interface 400 is configured to send data or any other messages to PIN 110. In one embodiment, PIN interface 400 includes a distributor, input arbiter, and an aggregator. The distributor and arbiter facilitate distributing data to one or more engines 402. The aggregator aggregates words that will be sent to nodes. When a request is received at PIN interface 400, PIN interface 400 determines which engine in engines 402 to send the request to.

[33]   In one embodiment, PIN interface 400 also provides a priority system for memory service requests. For example, one memory priority system may give a peek/poke memory service request the highest priority. Random read requests that are received with a fast track or higher priority indication are then given the next highest priority. All other requests are given a lowest priority. For example, random memory access requests are placed on a 132 entry first come first serve queue, DMA and PTP requests are placed on a single 64 entry first come first serve queue and these two queues are serviced on a round robin basis.

[34]   As shown, one or more engines 402 includes a peek/poke engine 410, a fast track engine 412, a random read/write engine 414, and a

PTP/DMA/RTI engine 416 according to one embodiment of the invention. Although these engines 402 are described, a person skilled in the art will appreciate that other engines may be provided to perform functions related to the memory access services. Engines 402 process a memory service request and provide the appropriate request to memory interface 404 to fulfill the memory service request. For example, engines 402 determine a memory address that data should be read from in memory 304 or the data and a memory address in which data should be written to in memory 304. The action is then performed according to a protocol associated with the memory service request.

[35]   Memory interface 404 receives memory service requests from memory interface 404 and provides them to SDRAM memory 406 and/or flash memory 408. Although SDRAM memory 406 and flash memory 408 are shown, it will be understood that a person skilled in the art will appreciate other memories that may be used.

[36]   The types of services that are provided by engines 402 will now be described.

[37]   When a peek memory service request is received at PIN interface 400, it determines that the request should be sent to peek/poke engine 410. The peek request is received in one or more data words and PIN interface 400 is configured to determine from data in the data words that a peek should be performed. The peek request is then forwarded to peek/poke engine 410, which determines peek address(es) that data should be read from. In one embodiment, peek requests are used to read data from memory or registers found in controller 302. For example, registers storing parameters 422 in ports 418 may be peeked. The data request at the determined address(es) is then sent to appropriate registers. The data is then returned to peek/poke engine 410 and sent to the requesting node through PIN interface 400 and PIN 110.

[38]   In order to provide flow control, the requesting node waits for receipt of prior peek data before initiating a new peek request to the same memory.

[39]   When a poke request is received at PIN interface 400, PIN interface 400 determines that the request should be sent to peek/poke engine 410. In one embodiment, a poke request is sent in one or more data words and PIN interface 400 determines from the one or more data words that the request should be sent to peek/poke engine 410. Peek/poke engine 410 receives a poke address word from the requester and a poke data word to write to the address previously supplied by the poke address word.   For example, registers including parameters 422 may have data written to them. Peek/poke engine 410 also determines from the one or more data words which register to write the data to.

[40]   After the data has been written, a poke acknowledgement may besent by peek/poke engine 410 to the requesting node through PIN 110 and PIN interface 400. Flow control can be realized by requiring a requesting node to wait for full acknowledgement before initiating a new poke to the same memory.

[41]   Fast track engine 412 is provided to perform memory access services that have a higher priority. Thus, fast track engine 412 allows requesting nodes to send requests and data in an expedited manner.

[42]   When a memory random access read or write is received at PIN interface 400, PIN interface 400 then provides the memory service request to random read/write engine 414. In one embodiment, a double word (32-bits) on a double word boundary may be read at a certain specified address or a burst read, which reads 16 double words on double word boundaries, may be performed.

[43]   In one embodiment, MRA read requests are placed in a queue and random read/write engine 414 services requests in a first in/first out methodology in one embodiment. When a request to memory 304 is ready, random read/write engine 414 sends the determined address with an indication of the appropriate memory that data should be read from to memory interface 404. The request is forwarded to memory 304 and data is read and returned to random read/write engine 414. The data can then be returned to the requesting node through PIN interface 400 and PIN 110.

[44] In order to maintain flow control, in one embodiment, the requesting node waits for receipt of prior MRA read data before initiating a new MRA read or write to the same memory. Thus, the requesting node may make a first read request to SDRAM memory 406 and a second request to flash memory 408 simultaneously but cannot make multiple requests to SRAM memory 406 or flash memory 408.

[45] When PIN 400 receives a MRA write request, it determines from one or more data words in the request that a MRA write should be performed. For example, a bit or any other indication may be set in the one or more data words to indicate the request is a MRA request. The request is then forwarded to random read/write engine 414, which determines a memory location from the one or more data words where the data should be written. The address is then stored in a table and when data for the write is received (either with the one or more data words containing the request or in one or more data words received later), the data is then stored in a temporary buffer. The MRA request is then placed in a queue. The queue is serviced in a first in/first out manner by random read/write engine 414.

[46] When the MRA write request is serviced, the data is retrieved from the temporary buffer and written to the address included in the appropriate entry of the random address queue. In this case, the data, address, and which memory to write the data are sent to memory interface 404, which writes the data to either SDRAM memory 406 or flash memory 408 at the address specified. Random read/write engine 414 then sends a MRA write acknowledgement to the requesting node. Flow control is maintained because a requesting node waits for a MRA write acknowledgement before issuing a new random MRA read or write to the same memory.

[47] A plurality of ports 418 are provided for the direct memory access (DMA), point-to-point (PTP), and real-time input (RTI) memory services. In one embodiment, each port includes DAG parameters and other parameters 422 and a temporary buffer 424. In a preferred embodiment the DAG is used to generate sequences of addresses for both reading and writing memory. For example, a

node that desires to access a pattern of memory locations obtains the addresses from the DAG. The DAG can be configured in various ways such as, e.g., by a control node poking port configuration parameters. Another way to configure the DAG is dynamically via PTP control words. Details of the DAG are provided in following sections.

[48]   One or more DAG parameters 422 associated with a port 148 are used by DAG 420 to determine the appropriate data to retrieve from memory 304, or the appropriate location in memory to update. Other parameters can be included, such as temporary buffer parameters, control and status register bits, producer information, consumer information, counts, and the like.

[49]

[50]   In one embodiment, each of ports 418 include a temporary buffer 424. Temporary buffer 424 is used to store one or more PTP/DMA/RTI words that are received from a requesting node. When data is stored in temporary buffer 424, an indication of what kind of request associated with the stored data is stored in queue 426.

[51]   A PTP_DMA_Queue 426 is maintained by thePTP/DMA/RTI engine 416 for servicing of ports. Various events as described below cause a port to be placed on this first-in-first-out queue.

[52]   The services provided by PTP/DMA/RTI engine 416 will now be described.

[53]   Direct memory access services include a DMA read and a DMA write service. In a DMA read service, any of the ports 418 can serve as a source of a DMA channel set up by a requesting node 301. When a DMA read request for a port i in ports 418 is serviced, DAG 420 is configured with the DAG parameters for port i. Data is then read from memory 304, such as SDRAM memory 406 or flash memory 408, using the just configured DAG 420 by PTP/DMA/RTI engine 416.

[54]   The DMA read may be for a large chunk of data and multiple reads may be needed to read the entire requested chunk of data. Thus, memory controller 302 may send multiple chunks of data to a requesting node 301 in

response to a DMA read. In one embodiment, counts are used to determine how much data to read. For example, chunks of data may be read in 32-bit words but the read request may be for seven bytes. The count would be set to seven and when the first word, which includes four bytes, is read, the count is decremented to three. When the next byte is read, the count is decremented to zero and only three bytes are read because the count was three. In some cases, multiple DMA reads may be serviced for a node.

[55] In order to maintain flow control, memory controller 302 waits for a DMA read chunk acknowledgment from the requesting node before transmitting the next chunk of data. Also, PTP/DMA/RTI engine 416 waits for a DMA done message from the requesting node until a new DMA read from the same memory 304, such as SDRAM memory 406 or flash memory 408, is initiated.

[56] PTP/DMA/RTI engine 416 can also perform a DMA write. Any of the ports in ports 418 may serve as the destination of a DMA channel set up by a requesting node. Temporary buffer 424 is provided in each of ports 418 in order to store incoming DMA data that is eventually written into memory 304. Although buffer 424 is described, it will be understood that buffer 424 may not be used and the data may be streamed to PTP/DMA/RTI engine 416. Because a DMA write might be a write for large amounts of data, the data may arrive in multiple data words over a period of time. When a DMA write request is received at a port i in ports 418, if port i's temporary buffers 424 are already full, an error message is sent to the requesting node. If not, the data is written sequentially into port i's temporary buffer 424 and a corresponding DMA write request is placed in queue 426. As more data is received on port i, the data is written sequentially into the port's temporary buffer 424 if it is not already full. When the last data word for the DMA write request is received on port i, a DMA write request is placed in queue 426. Although the above sequence is described, it will be understood that a person skilled in the art will appreciate other ways of handling the received data.

[57] When the DMA write request is ready to be serviced by PTP/DMA/RTI engine 416, DAG 420 of PTP/DMA/RTI engine 416 is configured

with DAG parameters 422 for port i. Each successive DMA write request is read from queue 426 and the corresponding data in port i's temporary buffer 424 is then written to memory 304, such as SDRAM memory 406 or flash memory 408, using the just configured DAG 420. DAG 420 may calculate addresses based on one or more parameters 422 associated with port I and an address associated with the applicable memory DMA request. The addresses may be calculated for each successive DMA write request and DAG 420 may be configured with parameters 422 for each write request.

[58]   In order to maintain flow control, the transmitting node waits for a chunk acknowledgment from memory controller 302 that indicates the chunk of data has been stored in temporary buffer 424 before transmitting the next chunk of data to be stored in port I's temporary buffer 424. Additionally, the requesting node waits for a DMA done message from memory controller 302 before initiating a new DMA write to the same memory 304.

[59]   In one embodiment, counts are used to determine how much data to write. For example, chunks of data may be received in 32-bit words. The write request may be for seven bytes. The count would be set to seven and when the first word, which includes four bytes, is received and written, the count is decremented to three. When the next word is received, the count is decremented to zero and only three bytes are written because the count was three.

[60]   Point-to-point memory services may also be performed by PTP/DMA/RTI engine 416. Nodes 301 may read and write memory 304 and update selected port parameters 422 via any of ports 418 using a point-to-point protocol. Memory controller 302 adheres to all point-to-point conventions, performs forward and backward ACKing, and also maintains counts for consumers and producers. Additionally, flow control is maintained because of the point-to-point conventions. For example, in a write request, neither temporary buffer 424 for ports 418 nor a buffer in memory 304 will overflow so long as the requesting node adheres to PTP conventions. Additionally, in a read

request, memory controller 302 will not overflow the consuming node's input buffer as long as the requesting node adheres to PTP conventions.

[61]   PTP/DMA/RTI engine 416 may perform point-to-point memory services using a number of modes. For example, an auto-source mode provides an infinite source of data. A read occurs automatically when there is available space in a consuming node's input buffer and read requests are not used. An infinite-sink mode may be provided to provide an infinite sink for data. In this case, a write occurs when there is data in temporary buffer 424 and new data overwrites old data when the main buffer is full. In one embodiment, memory 304 includes a main buffer where data is written to. Thus, data is read from temporary buffer 424 and written to the main buffer. Although a main buffer is described, it will be understood that data may be written to other structures in memory 304. A finite-sink mode provides a finite sink for data. In this case, a write occurs when there is data in temporary buffer 424 and available space in the main buffer and writing stops when the main buffer is full. A buffer mode implements a first in/first out (FIFO) queue. In this case, writes fill the main buffer while reads drain the main buffer. A write occurs when there is data in the temporary buffer and available space in the main buffer. A read occurs when there is sufficient data in the main buffer and available space in the consuming-nodes input buffer. A basic mode provides unrestricted writing to a data structure. In this case, a write occurs when there is data in the temporary buffer, and old data in memory is overwritten. Also, the basic mode provides unrestricted reading of a data structure. A read occurs after an explicit read request is received and there is available space in the consuming nodes input buffer.

[62]   Fig. 5 illustrates the general design of an engine such as PTP/DMA/RTI engine 416 of Fig. 4.

[63]   Data packets are received from a data source such as a distributor (e.g., from PIN Interface 400 of Fig. 4). The payload portion of each incoming packet together with a bit indicating whether the payload is a data word or control word is stored in port temporary buffer 600. In a preferred embodiment, packets

17

are 51 bits wide and can include destination information, control information, parameter information, data, or a combination of these types of information. When a port is serviced, control words and data words are read from port temporary buffer 600 and sent to control system 604 or unpacker 608, respectively.

Port parameters can be updated by information in "poke packets" or by control-word information in incoming PTP/DMA packets. The parameter update information is provided to parameter control system 602. Port parameters are used to define characteristics of a port for specific or desired functionality. For example, port parameters control characteristics of temporary buffers, removing control and data words from the temporary buffer for processing, unpacking data (double-) words into records in preparation for writing to main memory, writing and reading records to main memory, packing records read from memory into double-words and composing appropriate MIN words for transmission to the consumer node, sending various control words – forward and backward acknowledgements, DMA chunk acknowledgements and DMA Done messages – to the producer and consumer nodes; and other functions.

[64] Unpacked data produced by unpacker 608 can include one or more records. Each record can be 8, 16 or 32 bits. A 4-bit byte select is sent with each 32-bit unpacked datum to indicate which of the bytes contain valid data and are to be written to memory.

[65] Control words are used to specify parameters and other control information and are discussed in detail in the sections, below. For example, a control word can include information that indicates whether a parameter update is to be followed by a read using the updated port parameters.

[66] Data address generator 606 is used to generate an address, or addresses, for use in reading from or writing to memory. The data address generator is configured by the DAG parameters included in the port parameters 602. Packer 612 is used to pack records received from memory into 32-bit data words for transmission to the consuming node . Packet assembly 610 is used to

assemble the 32-bit data words into a standard PTP, DMA or RTI packets for transimission to the consuming node.

[67]   In a preferred embodiment, the XMC node adheres to the same network protocol conventions as other nodes in the ACE. Examples of ACE network protocols in the preferred embodiment include Peek/Poke, MRA, PTP, DMA, RTI, message, etc. This allows XMC nodes to benefit from the same scaling features and adaptable architecture of the overall system. Details of a network protocol used in the preferred embodiment can be found in the related patent application entitled "Uniform Interface for a Functional Node in an Adaptive Computing Engine," referenced above.

[68]   In a preferred embodiment of the XMC there are 64 ports – each one a combination input/output port . Three matrix interconnect network (MIN) (also referred to as the programmable interconnect network (PIN)) protocols – Direct-Memory-Access (DMA), Point-To-Point (PTP) and Real-Time-Input (RTI) – make use of these ports for both writing data to and reading data from memory.

[69]   Memory addresses for both writing and reading are generated by a logical DAG associated with each port. This logical DAG is actually a set of DAG parameters that are used to configure a single physical DAG, as needed, for memory writes and reads.

[70]   Each port also has a temporary buffer to temporarily store incoming PTP/RTI/DMA words from the MIN. The total size of all 64 temporary buffers is 16Kbytes arranged as 4K x 33 bit words. The 33rd bit of each word indicates whether a double-word is a data word or a control word, as described below.

[71]   Each XMC port is associated with a set of parameters that define the characteristics of that port. These parameters configure the XMC hardware when a port is called upon to perform one of the following tasks:

Writing incoming control and data words into the temporary buffer;

Removing control and data words from the temporary buffer for processing;

Unpacking data (double-) words into records in preparation for writing to main memory;

Writing records to main memory;

Reading records from main memory;

Packing records read from memory into double-words and composing appropriate MIN words for transmission to the consumer node; and

Sending various control words – forward and backward acknowledgements, DMA chunk acknowledgements and DMA Done messages – to the producer and consumer nodes.

[72] The value of each port parameter can be either static or dynamic. If static, then the parameter is updated only by a poke from the K-Node. If dynamic, then the parameter can be updated by a poke from the K-Node and also during normal XMC operation.

[73] The Control and Status Bits described in Table A are the parameters that direct the behavior of ports and define their mode of operation.

Parameter	Description
Port_Enabled (Static Value)	0: Port disabled
	1: Port enabled
Port_Type[1:0] (Static Value)	00: PTP
	01: PTP_Packet_Mode
	10: RTI
	11: DMA
Record_Size[1:0] (Static Value)	00: Byte (8-bit)
	01: Word (16-bit)
	10: Double-Word (32-bit)
DAG_Address_Mode[1:0] (Static Value)	00: 1-D
	01: 2-D
	10: Bit_Reverse
Auto_Read (Static Value)	0: Port does not support automatic reads
	1: Producer/Consumer counts can automatically trigger a read
	0: Consumer_Count not checked in auto read

Parameter	D scripti n
Buffer_Read (Static Value)	0: Consumer_Count not checked in auto read
	1: Consumer_Count >= 0 for auto read
Buffer_Write (Static Value)	0: New data overwrites old data in memory
	1: No writes to main-memory buffer when full
Update_Index (Static Value)	0: Update DAG_X_Index and DAG_Y_Index only by way of poke
	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap (Static Value)	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
	1: Start unpacking/packing new MIN word when DAG_Y_Index wraps
High_Speed_Write (Static Value)	0: Normal mode—port handles all incoming words
	1: High-speed mode—port does not support read requests
Burst (Static Value)	0: Normal DAG-addressing mode
	1: High-throughput mode for accessing contiguous blocks of 2-D data
Random_Access (Static Value)	0: Normal DAG addressing when performing read request
	1: DAG addressing bypassed when performing read request

[74]   The two DMA Bits in Table B are used to control DMA transfers from and to the XMC respectively.

Parameter	Description
DMA_Go (Static Value)	Poking this bit with a 1 initiates a DMA transfer from the XMC to Consumer
DMA_Write_Last_Word (Dynamic Value)	0: DMA_Write_Last_Word_Of_Chunk initiated DMA service request
	1: DMA_Write_Last_Word initiated DMA service request

Table B -- DMA Bits

[75] The DAG parameters in Table C – together with DAG_Address_Mode – determine the sequence of addresses generated by the port's Data Address Generator. See section 3.2 for more details.

Table C -- DAG Parameters

DAG_Origin[27:0] (Dynamic Value)	Unsigned integer; Units = bytes; Base address of block 1-D mode *read*: DAG Address = DAG_Origin + DAG_X_Index 1-D mode *write*: DAG Address = DAG_Origin + DAG_Y_Index 2-D mode: DAG Address = DAG_Origin + DAG_X_Index + DAG_Y_Index Bit_Reverse mode read or write: DAG Address = DAG_Origin + reverse(DAG_X_Index)† Must be on a Dword boundary, i.e. [1:0] = 00;
	†reverse(b, p0, p1) – reverse bits from bitpos p0 thru bitpos p1, e.g.: for (i = 0; i <=(p1-p0-1)/2; i++) { swap(b[p0+i], b[p1-i]); }

DAG_X_Index[27:0]
(Dynamic Value)
	Unsigned integer; Units = bytes
	Initial value must be less than DAG_X_Limit.
	
	1-D mode, after a <u>read</u>, or
	2-D mode, after a read or write:
	DAG_X_Index += DAG_X_Stride
	
	Bit_Reverse mode, after a read or write:
	DAG_X_Index += 1, 2, or 4 (byte, word, dword record, respectively)
	
	Then test:
	if DAG_X_Index ≥ DAG_X_Limit (+*X Wrap*)
	    DAG_X_Index -= DAG_X_Limit
	else if DAG_X_Index < 0 (-*X Wrap*)
	    DAG_X_Index += DAG_X_Limit

DAG_X_Stride[27:0] (Dynamic Value)	Signed integer; Units = bytes Absolute value must be less than DAG_X_Limit. 1-D, 2-D mode: Increment/decrement to DAG_X_Index Bit_Reverse mode: reverse(1) = 2^(n-1), i.e. a single bit marking the leftmost bit position to be reversed in DAG_X_Index
DAG_X_Limit [27:0] (Dynamic Value)	Unsigned integer; Units = bytes 1-D mode *read* – block size 1-D mode *write* – not used 2-D mode read or write – X block size Bit_Reverse mode – block size

`DAG_Y_Index[27:0]`
(Dynamic Value)
	Unsigned integer; Units = bytes
	Initial value must be less than `DAG_Y_Limit`.
	
	1-D mode, after a <u>write</u>, or ·
	2-D mode, after an X Wrap:
	　`DAG_Y_Index += DAG_Y_Stride`
	
	Bit_Reverse mode − not used
	
	Then test:
	　if `DAG_Y_Index` ≥ `DAG_Y_Limit` (+*Y Wrap*)
	　　`DAG_Y_Index -= DAG_Y_Limit`
	　else if `DAG_Y_Index` < 0 (-*Y Wrap*)
	　　`DAG_Y_Index += DAG_Y_Limit`

`DAG_Y_Stride[27:0]` (Dynamic Value)	Signed integer; Units = bytes Absolute value must be less than `DAG_Y_Limit` 1-D, 2-D mode: Increment/decrement to `DAG_Y_Index` Bit_Reverse mode − not used
`DAG_Y_Limit[27:0]` (Dynamic Value)	Unsigned integer; Units = bytes 1-D mode *read* − not used 1-D mode *write* − block size 2-D mode read or write − Y block size Bit_Reverse mode − not used

[76]   The Temporary-Buffer Parameters in Table D define the size of
temporary buffer of a port and provide the write-pointer and read-pointer needed
to implement a circular first-in-first-out queue.

Table D — Temporary-Buffer Parameters

Parameter	Description
Buffer_Size[3:0] (Static Value)	0000: 4 (bytes)
	0001: 8
	0010: 16
	0011: 32
	0100: 64
	0101: 128
	0110: 256
	0111: 512
	1000: 1024
	1001: 2048
	1010: 4096
	1011: 8192
	1100:16384
Write_Address[11:0] (Dynamic Value)	Write pointer
Read_Address[11:0] (Dynamic Value)	Read pointer

[77]   The Producer/Consumer Information in Table E is used in various fields in the MIN words that are sent to the Data Producer, Control Producer and Consumer.

Table E -- Producer/Consumer Information

Parameter	Description
Data_Producer_ID[7:0] (Static Value)	Address of Data Producer (The source of PTP/DMA data words)
Data_Producer_Mode (Static Value)	Mode bit of Data Producer
Data_Producer_Port[5:0] (Static Value)	Port number of Data Producer
Data_Producer_Task[4:0] (Static Value)	Task number of Data Producer
Control_Producer_ID[7:0] (Static Value)	Address of Control Producer (The source of PTP control words)
Control_Producer_Mode (Static Value)	Mode bit of Control Producer
Control_Producer_Port[5:0] (Static Value)	Port number of Control Producer
Control_Producer_Task[4:0] (Static Value)	Task number of Control Producer
Consumer_ID[7:0] (Static Value)	Address of Consumer (The destination of read data)
Consumer_Mode (Static Value)	Mode bit of Consumer
Consumer_Port[5:0] (Static Value)	Port number of Consumer
Consumer_Task[4:0] (Static Value)	Task number of Consumer

[78]   The Counts in Table F provide flow control between (a) the Data and Control Producers and the XMC, (b) the temporary buffer and the main-memory buffer (when Buffer_Write = 1) and (c) the XMC and the Consumer.

Table F -- Counts

Parameter	Description
ACK_Count[13:0] (Dynamic Value)	A signed number indicating the number of bytes in a port's temporary buffer minus 1; A port is serviced when ACK_Count ≥ 0 Initialized at system reset to −1 indicating that the temporary buffer is empty; and then incremented in response to forward ACKs from the Data and Control Producers indicating the number of data/control words, expressed in bytes, placed in the temporary buffer; and then decremented when the XMC sends backward ACKs to the Data Producer and Control Producer indicating the number of data words and control words, respectively − expressed in bytes − removed from the temporary buffer
Read_Count[13:0] (Static Value)	An unsigned number indicating the number of records read from memory and sent to the consumer node per read-request or auto-read
Producer_Count[13:0] (Dynamic Value)	A signed number reflecting the available space, in bytes, in the Consumer's input-buffer; Producer_Count < 0 indicates that the consumer node input-buffer has available space for Read_Count records Should be initialized to RC − CBS − 1 (a negative value), where RC is Read_Count, expressed in bytes, and CBS is the Consumer's input-buffer size, in bytes; Incremented when the XMC sends forward ACKs to the Consumer indicating the amount of data, in bytes, read from memory and sent to the Consumer; and then decremented in response to backward ACKs from the Consumer indicating the amount of space, in bytes, freed up in the Consumer's input buffer

Parameter	Description
Consumer_Count[27:0] (Dynamic Value)	A signed number reflecting the number of bytes in the main-memory buffer; `Consumer_Count` ≥ 0 indicates that the main-memory buffer has at least `Read_Count` records; Applicable only when `Buffer_Read = 1` Should be initialized to a (negative) value between `TBS - MBS and -RC`, where `TBS` is the temporary-buffer size, in bytes, `MBS` is the main-memory-buffer size, in bytes, and `RC` is `Read_Count`, expressed in bytes; Incremented when the XMC moves data from the temporary buffer to the main-memory buffer; and then decremented when the XMC sends forward ACKs to the Consumer indicating the amount of data, in bytes, read from the main-memory buffer and sent to the Consumer
Buffer_Full_Offset[27:0] (Static Value)	A signed number which, when added to `Consumer_Count`, indicates XMC buffer status; `Consumer_Count + Buffer_Full_Offset` ≥ 0 indicates that the main-memory buffer is full; The main-memory buffer is considered to be *full* when it does not have at least a temporary-buffer's worth of available space; Applicable only when `Buffer_Write = 1` Should be initialized to `TBS - MBS - ICC - 1` where `TBS` is the temporary-buffer size, in bytes, `MBS` is the main-memory-buffer size, in bytes, and `ICC` is the initial value of `Consumer_Count`

[79] Table C, above, describes XMC DAG parameters. The 3 accessing modes (1-D, 2-D, and Bit_Reverse) are explained below. Special cases are also discussed relating to Y-Wrap and Burst Mode.

[80] The DAG includes the ability to generate patterned addresses to memory. Three parameters -- Index, Stride, and Limit -- in each of X and Y define these patterns. In the simplest 1-dimensional case, the Index parameter is incremented by Stride, tested against the block size given by Limit, and then added to Origin to determine the final address.

[81] Note that Stride is a signed quantity, and can be negative to enable stepping backwards through a block of memory addresses. If the Index is incremented/decremented outside the block (0 thru Limit-1), the Limit is subtracted/added respectively to bring the address back within the block. In this

way, circular buffers with automatic wrap-around addressing are easily implemented. In general, any type of addressing, address incrementing/decrementing, indexing, etc., can be used with DAGs of different designs.

[82] In a 1-D addressing mode, the DAG writes or reads addresses in a linear fashion. On each advance, DAG_X_Stride is added to DAG_X_Index, and the result tested greater than or equal to DAG_X_Limit and less than 0 (since DAG_X_Stride can be negative). In these cases, DAG_X_Index is decremented or incremented, respectively, by DAG_X_Limit, thus restoring it to the proper range.

[83] When in 1-D Write Mode, only, the DAG uses the DAG_Y_Index, DAG_Y_Stride, and DAG_Y_Limit parameters, not X, to compute the write address. This is so that read operations can be performed concurrently, using the X parameters in the usual way, to create a circular buffer such as a FIFO.

[84] In a 2-D addressing mode, the DAG writes or reads addresses in 2-dimensional "scan-line" order, utilizing both the X and Y parameters similarly to the 1-D mode. X advance is performed first, and an X Wrap (either + or -) causes a Y advance (and thus a potential Y Wrap as well). See the DAG advance pseudo-code description in section 3.2.4 below.

[85] Note that Y parameters are always specified in units of bytes, not scan lines or data items.

[86] Bit-reversed addressing is included in the hardware to enable implemention of Fast Fourier Transforms and other interleaved or "butterfly" computations. In this mode, bits within the DAG_X_Index field are reversed (swapped) just prior to using them in the memory address computation.

[87] In Bit_Reverse mode, DAG_X_Stride is not used as an increment, but instead determines the range of bits to reverse within DAG_X_Index. Specifically, the DAG_X_Stride should be set to reverse(1) = $2^{(n-1)}$ = 1/2 the size of the block in bytes. Bits p through n-1 will be reversed in the DAG_X_Index, where p = 0, 1, 2 for Record_Size of byte, word, and dword, respectively.

[88]   Example: For a 2^12 = 4096-point FFT in byte mode, parameters might be

DAG_X_Index = 0x0, DAG_X_Stride = 0x800, DAG_X_Limit = 0x1000 .

Thus the hardware will reverse bits 0-11, and the address sequence is

address	reverse(address, 0, 11)
0	0x000
1	0x800
2	0x400
3	0xc00
4	0x200
5	0xa00
...	...

[89]   As in other modes, the resulting reversed DAG_X_Index value is added to the Origin address before being used to access memory.

[90]   In Bit_Reverse mode, note that the starting DAG_X_Index, the DAG_X_Limit, and the Origin are byte addresses specified normally – NOT bit-reversed. However, in this mode, the Origin must be on a double-word boundary, i.e. bits [1:0] = 00;

[91]   Although the X Wrap mechanism works in Bit_Reverse mode, typically DAG_X_Index is initialized to 0 and a single array of 2^n values will be addressed once.

[92]   Combining the above parameter definitions, the calculation of the DAG memory addresses is as follows:

[93]   When the DAG is advanced:

- If Address_Mode = 1-D and the DAG is generating a <u>Read</u> Address [or Bit_Reverse mode]:
  o DAG_X_Index = DAG_X_Index + DAG_X_Stride [+ 1, 2, or 4 instead if Bit_Reverse mode]
  o If DAG_X_Index >= DAG_X_Limit, *(+X wrap)*
    ▪ DAG_X_Index = DAG_X_Index – DAG_X_Limit;
  o Else if DAG_X_Index < 0, *(-X wrap)*
    ▪ DAG_X_Index = DAG_X_Index + DAG_X_Limit;

- o Memory Address = Origin + DAG_X_Index [+ reverse(DAG_X_Index) instead if Bit_Reverse mode]
- If Address_Mode = 1-D and the DAG is generating a <u>Write</u> Address:
    - o DAG_Y_Index = DAG_Y_Index + DAG_Y_Stride
    - o If DAG_Y_Index >= DAG_Y_Limit, *(+Y wrap)*
        - ▪ DAG_Y_Index = DAG_Y_Index – DAG_Y_Limit;
    - o Else if DAG_Y_Index < 0, *(-Y wrap)*
        - ▪ DAG_Y_Index = DAG_Y_Index + DAG_Y_Limit;
    - o Memory Address = Origin + DAG_Y_Index;
- If Address_Mode = 2-D:
    - o DAG_X_Index = DAG_X_Index + DAG_X_Stride;
    - o If DAG_X_Index >= DAG_X_Limit, *(+X wrap)*
        - ▪ DAG_X_Index = DAG_X_Index – DAG_X_Limit;
        - ▪ DAG_Y_Index = DAG_Y_Index + DAG_Y_Stride;
        - ▪ If DAG_Y_Index >= DAG_Y_Limit, *(+Y wrap)*
            - • DAG_Y_Index = DAG_Y_Index – DAG_Y_Limit;
        - ▪ Else if DAG_Y_Index < 0, *(-Y wrap)*
            - • DAG_Y_Index = DAG_Y_Index + DAG_Y_Limit;
    - o Else if DAG_X_Index < 0, *(-X wrap)*
        - ▪ DAG_X_Index = DAG_X_Index + DAG_X_Limit;
        - ▪ DAG_Y_Index = DAG_Y_Index + DAG_Y_Stride;
        - ▪ If DAG_Y_Index >= DAG_Y_Limit, *(+Y wrap)*
            - • DAG_Y_Index = DAG_Y_Index – DAG_Y_Limit;
        - ▪ Else if DAG_Y_Index < 0, *(-Y wrap)*
            - • DAG_Y_Index = DAG_Y_Index + DAG_Y_Limit;
    - o Memory Address = Origin + DAG_X_Index + DAG_Y_Index

[94]   Tables G-N, below, shows "for loop" representations in C pseudo-code of various DAG addressing modes. Capitalized names such as Origin, Index, Stride, Limit, etc. represent the corresponding DAG parameters. The examples below all assume Record_Size = Dword = 4 bytes, and positive

31

strides. Note that DAG parameters are always given in units of bytes, not records.

Table G — Linear Addressing Definiti n

```
// Linear addressing

void DAG_Linear (    byte *Origin,
                     uint28 Index,
                     int28 Stride,
                     uint28 Limit,
                     int28 count) {       // count < Limit

  int28 i;

  for (i=Index; i<Index+count; i+=Stride) {
    printf("%d: %d %d\n", i, Origin+i, (dword)Origin[i]);
  }
}
```

Table H -- Linear Addressing Example

```
Given the following memory contents,

        address         contents
        0x22bee8            7
        0x22bee4            6
        0x22bee0            5
        0x22bedc            4
        0x22bed8            3
        0x22bed4            2
        0x22bed0            1
        0x22becc            0

the function call

        DAG_Linear(0x22bed0, 0, 1*4, 20*4, 6);

yields

        iteration    address        contents
            0:       0x22bed0           1
            1:       0x22bed4           2
            2:       0x22bed8           3
            3:       0x22bedc           4
            4:       0x22bee0           5
            5:       0x22bee4           6
```

Table I -- Circular Addressing Definiti n

```
// Circular (wraparound) addressing

void DAG_Circular_1D(      byte *Origin,
                           uint28 Index,
                           int28 Stride,
                           uint28 Limit,
                           int28 count) {
  int28 i, imod;

  for (i=Index; i<Index+count; i+=Stride) {
    imod = i % Limit;
    printf("%d: %d %d\n", i, Origin+imod, (dword)Origin[imod]);
  }
}
```

Table J -- Circular Addressing Example

```
Given the following memory contents,

        address              contents
        0x22bee8                7
        0x22bee4                6
        0x22bee0                5
        0x22bedc                4
        0x22bed8                3
        0x22bed4                2
        0x22bed0                1
        0x22becc                0

the function call

        DAG_Circular_1D(0x22bed0, 0, 1*4, 6*4, 10);

yields

        iteration       address       contents
             0:         0x22bed0          1
             1:         0x22bed4          2
             2:         0x22bed8          3
             3:         0x22bedc          4
             4:         0x22bee0          5
             5:         0x22bee4          6
             6:         0x22bed0          1
             7:         0x22bed4          2
             8:         0x22bed8          3
             9:         0x22bee0          4
```

Table K — 2D Addressing Definition

```
// 2-D Addressing

void DAG_2D( byte *Origin,
             uint28 xIndex,
             int28 xStride,
             uint28 xLimit,
             uint28 yIndex,
             int28 yStride,
             uint28 yLimit) {
  int28 x, y;

  // Access a one-dimensional array through two loops (2-D)
  for (y=yIndex; y<yIndex+yLimit; y+=yStride) {
    for (x=xIndex; x<xIndex+xLimit; x+=xStride) {
      printf("%d %d: $d %d\n", x, y, Origin+x+y, (dword)Origin[x+y]);
    }
  }
}
```

Table L — 2D Addressing Example

Given the following memory contents
(a 2-D image, X x Y = 3 columns x 3 rows embedded in 5 columns x 4 rows),

address	contents
0x22bf18	9
0x22bf14	8
0x22bf10	7
0x22bf0c	6
0x22bf08	5
0x22bf04	4
0x22bf00	3
0x22befc	2
0x22bef8	1
0x22bef4	0

address	contents
0x22bf40	19
0x22bf3c	18
0x22bf38	17
0x22bf34	16
0x22bf30	15
0x22bf2c	14
0x22bf28	13
0x22bf24	12
0x22bf20	11
0x22bf1c	10

the function call

```
    DAG_2D(0x22bef8, 0, 1*4, 3*4, 0, 5*4, 15*4);
```

yields

```
    x    y      address        contents
    0    0:     0x22bef8           1
    4    0:     0x22befc           2
    8    0:     0x22bf00           3
    0   20:     0x22bf0c           6
    4   20:     0x22bf10           7
    8   20:     0x22bf14           8
    0   40:     0x22bf20          11
    4   40:     0x22bf24          12
    8   40:     0x22bf28          13  .
```

Table M -- Bit-Reverse Addressing Definition

```
// Bit-Reverse addressing (with wraparound)

void DAG_BitReverse(byte *Origin,
                    uint28 Index,
                    int28 Stride,
                    uint28 Limit,
                    int28 count) {
  int28 i, irev;

  for (i=Index; i<Index+count*4; i+=4) {// inc by 4 for dwords
     irev = Bit_Rev(i % Limit);          // swap bits 2 thru Stride bit
     printf("%d: %d %d\n", i, Origin+irev, (dword)Origin[irev]);
  }
}
```

Table N -- Bit-Reverse Addressing Example

```
Given the following memory contents (an 8-element block),

    address          contents
    0x22bef0            9
    0x22beec            8
    0x22bee8            7
    0x22bee4            6
    0x22bee0            5
    0x22bedc            4
    0x22bed8            3
    0x22bed4            2
    0x22bed0            1
    0x22becc            0

the function call

    DAG_BitReverse(0x22bed0, 0, 4*4, 8*4, 12);     // Stride = 2^(n-1) = 4

yields

    iteration     address        contents
        0:        0x22bed0           1
        4:        0x22bee0           5
        8:        0x22bed8           3
       12:        0x22bee8           7
       16:        0x22bed4           2
       20:        0x22bee4           6
       24:        0x22bedc           4
```

```
28:        0x22beec        8
32:        0x22bed0        1
36:        0x22bee0        5
40:        0x22bed8        3
44:        0x22bee8        7   .
```

[95]   Any of the 64 PTP/DMA ports can serve as the source of a DMA channel set up by the K-Node/Host. In a preferred embodiment, only one DMA channel to/from memory at a time can be supported.

Actions

When **Status_Register[i].DMA_Go** is poked with a 1,

1) Place a Service Request for Port i in the PTP_DMA_Queue if one is not already pending

When a **Service Request** for Port i is serviced with

Control_Register[i].Port_Type = DMA and Register[i].DMA_Go = 1:

1) Pop Port i from the PTP_DMA_Queue

2) If Status_Register[i].Port_Enabled = 0

    a) Send a **Port Disable Acknowledgement** to the K-Node

    b) Terminate servicing of Port i

3) Load Port-i DAG parameters into corresponding DAG registers

4) Note: When DAG_Address_Mode[i] = 1-D, the DAG uses the three X registers for reading and the three Y registers for writing

5) Read Read_Count[i] records from main memory under DAG direction, pack them from right to left[†] into double-words and send to Consumer[i] via a sequence of **DMA Read Data's** followed by a single **DMA Read Last Word**.

Flow Control

The K-Node waits for a **DMA Done** message from the destination node before initiating a new DMA read/write from/to the same memory.

DIRECT-MEMORY-ACCESS WRITE

Any of the PTP/DMA 64 ports can serve as the destination of a DMA channel set up by the K-Node/Host.

Actions

When a **DMA Write** from the MIN is received on Port i:

1) Place the 32-bit payload, together with a bit indicating that the double-word is a data word, sequentially into Port i's (33-bit-wide, circular) temporary buffer.

2) Increment Ack_Count[i] by 4.

When a **DMA Write Last Word Of Chunk** from the MIN is received on Port i:

1) Perform **DMA Write** actions.

2) Set Status_Register[i].DMA_Write_Last_Word to 0.

3) Place a Service Request for Port i in the PTP_DMA_Queue if one is not already pending.

When a **DMA Write Last Word** from the MIN is received on Port i:

1) Perform **DMA Write** actions.

2) Set Status_Register[i].DMA_Write_Last_Word to 1.

3) Place a Service Request for Port i in the PTP_DMA_Queue if one is not already pending.

When a **Service Request** for DMA-Port i is serviced

1) Pop Port i from the PTP_DMA_Queue

2) If Status_Register[i].Port_Enabled = 0

   a) Send a **Port Disable Acknowledgement** to the K-Node

   b) Terminate servicing of Port i

3) Load Port-i DAG parameters into corresponding DAG registers

4) Note: When DAG_Address_Mode[i] = 1-D, the DAG uses the three X registers for reading and the three Y registers for writing

5) Initialize signed-integer C to Ack_Count[i] / 4

6) While C >= 0:

   a) Decrement C by 1

   b) Remove double-word from temporary buffer

   c) Unpack double-word from right to left[†] and write records to memory under DAG direction.

7) Decrement Ack_Count[i] by 4 times the total number of double-words removed from Port i's temporary buffer

8) If Status_Register[i].DMA_Write_Last_Word = 0, send a **DMA Chunk Acknowledgement** to Data_Producer[i]; Omit if no records were written to memory

9) Else if Status_Register[i].DMA_Write_Last_Word = 1, send a **DMA Done** message to the K-Node; Omit if no records were written to memory

10) If Update_Index[i] = 1:

   a) Update X_Index[i] DAG parameter with X_Index DAG register

---

[†] Records are packed and unpacked from right to left because the XMC is *little endian*.

37

b) Update Y_Index[i] DAG parameter with Y_Index DAG register

[96]. The DMA source waits for a DMA Chunk Acknowledgement from the memory controller before transmitting the next chunk (chunk size must be less than or equal to the size of the port's temporary buffer).

[97] The K-Node waits for DMA Done message from the memory controller before initiating a new DMA read/write from/to the same memory.

[98] Nodes may read and write memory and update selected port parameters via any of the 64 ports of the memory controller using the point-to-point protocol. The memory controller performs forward and backward ACKing and maintains Consumer_Counts and Producer_Counts.

[99] The memory controller recognizes a data word where the payload field contains data to be written to memory and a control word where the payload field contains port-update information and a bit indicating whether the update is to be followed by a read using the DAG. When the update is followed by a read request the control word is called a Read Request. Table I, below, shows different types of control words. PTP data words and PTP control words may be sent to a memory Port in any order and are processed in the order received.

Field	Description
Payload[27:0]	New Parameter Value
Payload[30:28]	000: Update DAG_Origin
	001: Update DAG_X_Index
	010: Update DAG_X_Stride
	011: Update DAG_X_Limit
	100: Update DAG_Y_Index
	101: Update DAG_Y_Stride
	110: Update DAG_Y_Limit
	111: Update Read_Count
Payload[31]	0: No Read Request
	1: Read Request

Table I. PTP Control-Word Fields

[100] Generally, data words and control words sent to the XMC are generated independently by separate tasks running on separate nodes. Therefore,

when the XMC sends acknowledgements to the nodes to indicate that a control word or other message or information has been received, the XMC must send separate acknowledgments, with appropriate values, to the task or node that is producing data words. The task or node that is producing the data word is referred to as the "Data Producer". A task or node that is producing control words is referred to as the "Control Producer." The XMC maintains information on the Data Producer and Control Producer in order to properly send backward acknowledgements to both.

[101] In general, tasks or nodes can be referred to as a "process" or as a component that performs processing. Although specific reference may be made to hardware or software components, it should be apparent that functions described herein may be performed by hardware, software or a combination of hardware and software.

[102] In a preferred embodiment, all words – both data and control – arriving at a PTP/RTI port on the XMC are placed sequentially into the same temporary buffer. For a case where two types of words are generated independently, typically by different nodes, it is necessary to allocate a portion of the temporary buffer to data words and a portion to control words to prevent buffer overflow.

[103] When a PTP Write, PTP Packet-Mode Write or RTI Write from the MIN is received on Port i the following actions are performed:

1) Place the 32-bit payload, together with a bit indicating whether the word is a data word or control word, sequentially into Port i's (33-bit-wide, circular) temporary buffer.

When a **Forward Acknowledgement** from the MIN is received on Port i:

1) Increment Ack_Count[i] by Ack Value (which is positive) (Note: **Forward Acknowledgement's** from the Data_Producer and the Control_Producer are treated identically.)

2) Place a Service Request for Port i in the PTP_DMA_Queue if one is not already pending

When a **Backward Acknowledgement** from the MIN is received on Port i,

1) Increment Producer_Count[i] by Ack Value (which is negative)

2) If the sign bit of Producer_Count[i] is now a 1 (Producer_Count[i] is negative), place a Service Request for Port i in the PTP_DMA_Queue if one is not already pending

When a **Service Request** for PTP/RTI-Port i is serviced:

1) Pop Port i from the PTP_DMA_Queue

2) If Status_Register[i].Port_Enabled = 0

a) Send a **Port Disable Acknowledgement** to the K-Node

b) Terminate servicing of Port i

3) Load Port-i DAG parameters into corresponding DAG registers

4) Note: When DAG_Address_Mode[i] = 1-D, the DAG uses the three X registers for reading and the three Y registers for writing

5) If ((Control_Register[i].Write_Port = 1)   OR (Producer_Count[i] < 0)   ) AND
   (Control_Register[i].Buffer_Write = 0) OR the main buffer is NOT full)

a) Initialize signed-integer C to Ack_Count[i] / 4

b) While C >= 0:

i) Decrement C by 1

ii) Remove double-word from temporary buffer

iii) If the double-word is a data word:

(1) Unpack data word from right to left[†] and write records to memory under DAG direction.

iv) Else (if the double-word is a control word):

(1) Update indicated DAG Parameter

(2) If a read is indicated

(a) Read Read_Count[i] records from memory under DAG direction, pack them from right to left into double-words and send to Consumer[i] via a sequence of **PTP Read Data's**

(b) Break from While loop

c) Decrement Ack_Count[i] by 4 times the total number of data and control double-words removed from Port i's temporary buffer

6) Increment Consumer_Count[i] by 4 times the total number of data double-words removed from the Port i's temporary buffer and written to memory

7) Send a **Backward Acknowledgement** to Data_Producer[i] with an ACK value equal to minus 4 times the number of data words removed from Port i's temporary buffer; Omit if Control_Register[i].Port_Type = RTI or if no data words were consumed

8) Send a **Backward Acknowledgement** to Control_Producer[i] with an ACK value equal to minus 4 times the number of control words

removed from Port i's temporary buffer; Omit if no control words were consumed

9) If Control_Register[i].Auto_Read = 1 AND Producer_Count[i] < 0 AND (Control_Register[i].Buffer_Read = 0 OR Consumer_Count[i] >= 0)

a) Read Read_Count[i] records from memory under DAG direction, pack them from right to left into double-words and send to Consumer[i] via a sequence of **PTP Read Data's**

10) Increment Producer_Count[i] by 4 times the total number of double-words sent to Consumer[i] (via read requests *and* auto reads)

11) Decrement Consumer_Count[i] by 4 times the total number of double-words sent to Consumer[i] (via read requests *and* auto reads)

12) Send a **Forward Acknowledgement** to Consumer[i] with an ACK value equal to 4 times the number of double-words sent to Consumer[i] (via read requests *and* auto reads); Omit if no words were sent to Consumer[i]

13) If Update_Index[i] = 1:

a) Update X_Index[i] DAG parameter with X_Index DAG register

b) Update Y_Index[i] DAG parameter with Y_Index DAG register

14) Push a Service Request for Port i onto the PTP_DMA_Queue if one is not already pending.

XMC Modes

[104] In a preferred embodiment the XMC operates in eight basic modes. . These include the following:

[105] Basic Mode – Provides unrestricted reading of and writing to a data structure. A write occurs when there is data in the temporary buffer and old data overwritten. A read occurs after an explicit read request has been received and there is available space in the input buffer consuming node. It does not consume data.

[106] High-Speed-Write Mode – Similar to Basic Mode with the exception that read requests are not supported, thereby achieving higher throughput in writing to memory.

[107] Finite-Sink Mode – Provides finite sink for data. A write occurs when there is data in the temporary buffer and available space in the main buffer. Writing stops when the main buffer is full.

[108] Auto-Source Mode – Provides an infinite source of data. A read occurs automatically when there is available space in the input buffer of the consuming node. Read Requests are not used.

41

[109] Buffer Mode – Implements a buffer/FIFO. Writes fill the main buffer while reads drain the main buffer. A write occurs when there is data in the temporary buffer and available space in the main buffer. A read occurs when there is sufficient data in the main buffer and available space in the consuming node's input buffer.

[110] Y-Wrap Mode – Permits a write to memory to end in the middle of a double-word for the case when Record_Size is either byte or (16-bit) word.

[111] Burst Mode – A special high-throughput mode for reading and writing 2-D blocks of bytes. Similar to Y-Wrap Mode in that writes to memory can end in the middle of a double-word.

[112] Burst-Write Mode – Identical to Burst Mode except that – like High-Speed-Write Mode – read requests are not permitted. Achieves higher throughput than Burst Mode in writing to memory.

Basic Mode

[113] Basic Mode supports writing to and reading from memory with no restrictions on Port_Type, DAG parameters or the use of PTP control words. Reads are initiated either by a read request when Port_Type is PTP, PTP_Packet_Mode or RTI or by poking a 1 into DMA_Go when Port_Type is DMA.

[114] Table II lists the Control and Status Bit parameters that define Basic Mode.

42

Parameter	Description
Auto_Read	0: Port does not support automatic reads
Buffer_Read	0: Consumer_Count not checked in auto read
Buffer_Write	0: New data overwrites old data in memory
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
High_Speed_Write	0: Normal mode; The port handles all incoming words
Burst	0: Normal DAG-addressing mode
Random_Access	0: Normal DAG addressing when performing read request

Table II Settings for Basic Mode

Where:

1. The compound condition (ACK_Count >= 0 AND Producer_Count < 0) triggers the processing of words in the temporary buffer. ACK_Count ≥ 0 indicates that there are words in the temporary buffer. Producer_Count < 0 indicates that there is space available in the consumer's input buffer in the event that a read request is encountered.

2. Once processing begins, it continues until either a read request is encountered (and processed) or the entire contents of the temporary buffer – as indicated by ACK_Count when processing begins – has been dispatched.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

4. When a control word without a read is encountered, the indicated update is performed.

5. When a read request is encountered, the indicated update is performed and Read_Count records are then read from main memory under DAG direction, packed from right to left into double-words and sent to the consumer node.

6. Upon completion of processing:

   a) ACK_Count is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

   b) Consumer_Count is incremented by 4 x the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with an value equal to minus 4 x the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with an value equal to minus 4 x the total number of *control* words – if any – that are processed

   e) If a read request has been processed:

43

  i. `Producer_Count` is incremented by 4 x the number of double-words sent to Consumer

  ii. A **Forward Acknowledgement** is sent to Consumer with an value equal to 4 x the number of double-words sent to Consumer

 f) The port is placed back on the PTP/DMA service queue to process any remaining words in the temporary buffer

7. When a port is restricted to just writing – for example, when the port is a DMA sink – High-Speed-Write Mode is recommended due to its higher performance and because it is does not require `Producer_Count < 0` in order to process words from the temporary buffer.

High-Speed-Write Mode

[115] High-Speed-Write Mode is similar to Basic Mode with the exception that read requests are not supported. This can allows advantages such as not requiring that Producer_Count < 0 before words are removed from the temporary buffer is eliminated. Also, words can be removed from the temporary buffer at a higher rate.

[116] Table III lists the Control and Status Bit parameters that define High-Speed-Write Mode.

Parameter	Description
`Auto_Read`	0: Port does not support automatic reads
`Buffer_Read`	0: `Consumer_Count` not checked in auto read
`Buffer_Write`	0: New data overwrites old data in memory
`Update_Index`	1: Update `DAG_X_Index` and `DAG_Y_Index` after each DAG use
`New_MIN_Word_On_YWrap`	0: Ignore `DAG_Y_Index` wrap when unpacking/packing MIN words
`High_Speed_Write`	1: High-speed mode; the port does not support read requests
`Burst`	0: Normal DAG-addressing mode
`Random_Access`	0: Normal DAG addressing when performing read request

Table III Parameters for High-Speed-Write Mode

Where:

1. `ACK_Count ≥ 0`, indicating that there are words in the temporary buffer, triggers the processing of those words.

2. Once processing begins, the entire contents of the temporary buffer – as indicated by `ACK_Count` when processing begins – is processed.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

4. When a control word is encountered, the indicated update is performed.

5. Upon completion of processing:

   a) `ACK_Count` is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

   b) `Consumer_Count` is incremented by 4 x the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed

6. High-Speed-Write Mode is the recommended mode when a port is a DMA sink.

Finite-Sink Mode

[117] Finite-Sink mode allows data to be written to memory and preserved from being overwritten by subsequent data. This is useful, for example, for storing statistics data, an error log, etc. Table IV lists the Control and Status Bit parameters that define Finite-Sink Mode.

Parameter	Description
Auto_Read	0: Port does not support automatic reads
Buffer_Read	0: Consumer_Count not checked in auto read
Buffer_Write	1: No writes to main-memory buffer when full
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
High_Speed_Write	1: High-speed mode; Port does not support read requests
Burst	0: Normal DAG-addressing mode
Random_Access	0: Normal DAG addressing when performing read request

Table IV Parameters for Finite-Sink Mode

Where:

1. The compound condition (`ACK_Count >= 0` AND `Consumer_Count + Buffer_Full_Offset < 0`) triggers the processing of words in the temporary

45

buffer. $ACK\_Count \geq 0$ indicates that there are words in the temporary buffer. $Consumer\_Count + Buffer\_Full\_Offset < 0$ indicates that there is at least a temporary-buffer's worth of available space in the main-memory buffer.

2. Once processing begins, the entire contents of the temporary buffer – as indicated by ACK_Count when processing begins – is processed.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. There *is* flow control between the temporary buffer and main memory and so new data does *not* overwrite old.

4. When a control word is encountered, the indicated update is performed.

5. Upon completion of processing:

   a) ACK_Count is decremented by 4 X the total number of words – both data and control – removed from the temporary buffer

   b) $Consumer\_Count$ is incremented by 4 X the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 X the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 X the total number of *control* words – if any – that are processed

6. Once $Consumer\_Count + Buffer\_Full\_Offset \geq 0$, all processing of words from the temporary buffer stops and any remaining words in the temporary buffer remain there.

Auto-Source Mode

[118] An application may need to make use of tables of constants. For example, wave tables, pseudo-random data, etc., are typically written at system initialization and accessed in a continuous stream during real-time operation. Auto-Source Mode provides a means for accessing such data. Table V lists the Control and Status Bit parameters that define Auto-Source Mode.

Parameter	Description

46

Auto_Read	1: Producer/Consumer counts can automatically trigger a read
Buffer_Read	0: Consumer_Count not checked in auto read
Buffer_Write	0: New data overwrites old data in memory
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
High_Speed_Write	1: High-speed mode; The port does not support read requests
Burst	0: Normal DAG-addressing mode
Random_Access	0: Normal DAG addressing when performing read request

Table V Parameters for Auto-Source Mode

Where:

1. Whenever Producer_Count < 0, Read_Count records are read from main memory under DAG direction, packed from right to left[†] into double-words and sent to Consumer. After each auto read:

   a) Producer_Count is incremented by 4 x the number of double-words sent to Consumer

   b) A **Forward Acknowledgement** is sent to Consumer with a value equal to 4 x the number of double-words sent to Consumer

2. ACK_Count ≥ 0, indicating that there are words in the temporary buffer, triggers the processing of those words.

3. Once processing begins, the entire contents of the temporary buffer – as indicated by ACK_Count when processing begins – is processed.

4. Data words from the temporary buffer are unpacked from right to left and the records written to main-memory under DAG direction. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

5. When a control word is encountered, the indicated update is performed.

6. Upon completion of processing:

   a) ACK_Count is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

   b) Consumer_Count is incremented by 4 x the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed.

47

Buffer Mode

[119]   In a preferred embodiment, a port in Buffer Mode implements a first-in-first-out queue. A delay line – a queue in which the amount of data in the queue remains above a threshold – is a form of FIFO and can also be implemented in Buffer Mode. Table VI lists the Control and Status Bit parameters that define Buffer Mode.

Parameter	Description
Auto_Read	1: Producer/Consumer counts can automatically trigger a read
Buffer_Read	1: Consumer_Count >= 0 for auto read
Buffer_Write	1: No writes to main-memory buffer when full
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
High_Speed_Write	1: High-speed mode; The port does not support read requests
Burst	0: Normal DAG-addressing mode
Random_Access	0: Normal DAG addressing when performing read request

Table VI Parameters for Buffer Mode

Where:

1. The compound condition (ACK_Count >= 0 AND Consumer_Count + Buffer_Full_Offset < 0) triggers the processing of words in the temporary buffer. ACK_Count ≥ 0 indicates that there are words in the temporary buffer. Consumer_Count + Buffer_Full_Offset < 0 indicates that there is at least a temporary-buffer's worth of available space in the main-memory buffer.

2. Once processing begins, the entire contents of the temporary buffer – as indicated by ACK_Count when processing begins – is processed.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. There *is* flow control between the temporary buffer and main memory and so new data does *not* overwrite old.

4. When a control word is encountered, the indicated update is performed.

5. When processing of words from the temporary buffer is completed:

a) `ACK_Count` is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

b) `Consumer_Count` is incremented by 4 x the total number of data words written to main memory

c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed

7. The compound condition (`Consumer_Count >= 0` AND `Producer_Count < 0`) triggers an auto read in which `Read_Count` records are read from main memory under DAG direction, packed from right to left into double-words and sent to Consumer. After each auto read:

a) `Consumer_Count` is decremented by 4 x the number of double-words removed from the main memory buffer

b) `Producer_Count` is incremented by 4 x the number of double-words sent to Consumer

c) A **Forward Acknowledgement** is sent to Consumer with a value equal to 4 x the number of double-words sent to Consumer

8. The initial value of `Consumer_Count` sets a threshold on the amount of data in the main-memory buffer necessary for an auto read to occur. If the initial value of `Consumer_Count` is –n, then n is the amount of data, expressed in bytes, necessary for an auto read to occur.

9. The minimum number of double-words in the main-memory buffer – after an initial transient phase when the buffer is filling up – is: –((Initial value of `Consumer_Count`)/4 + `Read_Count`) double-words

10. For example, if the initial value of `Consumer_Count` is –40,000 (bytes) and `Read_Count` is 100 (double-words) then an auto read occurs only after 10,000 double-words (40,000 bytes) have been written into the main-memory buffer. When an auto read does occur, 100 double-words are removed from the buffer and `Consumer_Count` is decremented by 400 (bytes). Since there must have been at least 10,000 double-words in buffer before the auto read occurred, there must be at least 10,000 – 100 = 9,900 double-words in the buffer after the auto read occurred. This number, 9,900, is the minimum number of double-words that can be in the main-memory buffer after the initial transient when the buffer is filling up.

Y-Wrap Mode

[120] Y-Wrap Mode, along with Burst Mode and Burst-Write Mode, permit a write to memory to end in the middle of a double-word. Y-Wrap Mode can be used, for example, when writing a block of pixels (bytes) by rows into a two-dimensional frame buffer. In this case, the Y Wrap occurs when the last pixel of the block is

written into memory. Any remaining bytes in the last data word are discarded and the next block of pixels begins with a new data word from the MIN. Table VII lists the Control and Status Bit parameters that define Y-Wrap Mode.

Parameter	Description
Auto_Read	0: Port does not support automatic reads
Buffer_Read	0: Consumer_Count not checked in auto read
Buffer_Write	0: New data overwrites old data in memory
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	1: Start unpacking/packing new MIN word when DAG_Y_Index wraps
High_Speed_Write	1: High-speed mode; The port does not support read requests
Burst	0: Normal DAG-addressing mode
Random_Access	0: Normal DAG addressing when performing read request

5
Table VII Parameters for Y-Wrap Mode

Where:

1. ACK_Count $\geq$ 0, indicating that there are words in the temporary buffer, triggers the processing of those words.

10
2. Once processing begins, the entire contents of the temporary buffer – as indicated by ACK_Count when processing begins – is processed.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. Upon a Y Wrap (DAG_Y_Index wraps around), writing is immediately terminated and any remaining records in the
15
data (double-) word are discarded.

4. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

5. When a control word is encountered in the temporary buffer, the indicated update is performed.

20
6. When processing of words from the temporary buffer is completed:

a) ACK_Count is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

b) Consumer_Count is incremented by 4 x the total number of data words written to main memory

25
c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

---

[†] Records are packed and unpacked from right to left because the XMC is *little endian*.

50

d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed

7. Example: Suppose `Record_Size` = *byte*, `DAG_Address_Mode` = 2-D and the DAG is configured to address a 9x9 block of records. When the 21st double-word of an incoming block is encountered, only the right-most byte – which is the 81st byte of the block – is written to memory because `DAG_Y_Index` wraps immediately after that byte is written. The remaining three bytes in the double-word are discarded and writing of the next block begins with a new double-word from the MIN.

5

Burst Mode

[121]   Burst Mode can be useful in imaging or video applications (e.g.,
MPEG4, HDTV, etc.) that have high bandwidth/throughput requirements. In a
5   preferred embodiment, Burst Mode makes use of the Double Data Rate (DDR)
feature of DDR DRAM. Other applications can use other types of memory and need
not use the DDR feature. Burst Mode allows blocks of pixels to be either written to
or read from memory at very high rates. Burst Mode terminates writing (and reading)
of a double-word on an X-Wrap. This difference means that each line, not just each
10   block, begins with a new double-word. Table VIII lists the Control and Status Bit
parameters that define Burst Mode.

Parameter	Description
Auto_Read	0: Port does not support automatic reads
Buffer_Read	0: Consumer_Count not checked in auto read
Buffer_Write	0: New data overwrites old data in memory
Update_Index	1: Update DAG_X_Index and DAG_Y_Index after each DAG use
New_MIN_Word_On_YWrap	0: Ignore DAG_Y_Index wrap when unpacking/packing MIN words
High_Speed_Write	0: Normal mode; The port handles all incoming words
Burst	1: High-throughput mode for accessing contiguous blocks of 2-D data
Random_Access	0: Normal DAG addressing when performing read request

Table VIII Parameters for Burst Mode

15   Where:

1. The compound condition (ACK_Count >= 0 AND Producer_Count < 0)
   triggers the processing of words in the temporary buffer. ACK_Count ≥ 0
   indicates that there are words in the temporary buffer. Producer_Count < 0
   indicates that there is space available in the consumer's input buffer in the event that a
20   read request is encountered.

2. Once processing begins, it continues until either a read request is encountered (and
   processed) or the entire contents of the temporary buffer – as indicated by
   ACK_Count when processing begins – has been dispatched.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records
25   written to main-memory under DAG direction. Upon an X Wrap (DAG_X_Index

---

[†] Records are packed and unpacked from right to left because the XMC is *little endian*.

52

wraps around), writing is immediately terminated and any remaining records in the data (double-) word are discarded.

4. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

5. When a control word without a read is encountered, the indicated update is performed.

6. When a read request is encountered, the indicated update is performed and Read_Count records are then read from main memory under DAG direction, packed from right to left into double-words and sent to the consumer node.

7. Upon completion of processing:

   a) ACK_Count is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

   b) Consumer_Count is incremented by 4 x the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed

   e) If a read request has been processed:

      i. Producer_Count is incremented by 4 x the number of double-words sent to Consumer

      ii. A **Forward Acknowledgement** is sent to Consumer with a value equal to 4 x the number of double-words sent to Consumer

   f) The port is placed back on the PTP/DMA service queue to process any remaining words in the temporary buffer

8. In the restriction above – *The Data Producer ACKs in multiples of 4 x ceil(DAG_X_Limit/4)* – ceil(DAG_X_Limit/4) is the number of double-words needed for each line in a block of pixels. 4 x ceil(DAG_X_Limit/4) is that number converted to bytes. The restriction guarantees that ACK_Count will always reflect an integral number of lines in the temporary buffer and the port will therefore always write an integral number of lines to memory.

9. The restriction above – *Read_Count is an integer multiple of DAG_X_Limit* – guarantees that the port will always read an integral number of lines from memory.

10. Write example: Suppose Record_Size = *byte*, DAG_Address_Mode = 2-D and the DAG is configured to address a 9x9 block of records. When the 3rd, 6th, 9th, 12th, 15th, 18th, 21st, 24th or 27th double-word of an incoming block is encountered, only the right-most byte – which is the 9th byte of a line – is written to memory because DAG_X_Index wraps immediately after that byte is written. The remaining three bytes in the double-word are discarded and writing of the next line in the block begins with a new double-word from the MIN. Notice that this incoming 9x9 block of pixels requires 27 double-words in Burst Mode, but only 21 double-words in Y-Wrap Mode.

11. Read example: Suppose `Record_Size` = *byte*, `DAG_Address_Mode` = 2-D, the DAG is configured to address a 9x9 block of records and Read_Count = 81. Now suppose that a read request is encountered in the temporary buffer. The port will read bytes from memory and pack them into outgoing double-words. But when the port gets to the 3rd, 6th, 9th, 12th, 15th, 18th, 21st, 24th or 27th double-word, it will place only a single byte – the 9th byte of a line – in the double-word (in the right-most position) because `DAG_X_Index` wraps immediately after that byte is read. The next byte – the first byte of the next line – goes into a new double-word. Notice that this outgoing 9x9 block of pixels requires 27 double-words in Burst Mode, but only 21 double-words in Basic Mode.

Burst-Write Mode

[122]   Burst-Write Mode allows higher throughput than Burst Mode by not supporting read requests and by not requiring Producer_Count < 0 in order to begin processing words from the temporary buffer. Table IX lists the Control and Status Bit parameters that define Burst-Write Mode.

Parameter	Description
Auto_Read	0: Port does not support automatic reads
Buffer_Read	0: `Consumer_Count` not checked in auto read
Buffer_Write	0: New data overwrites old data in memory
Update_Index	1: Update `DAG_X_Index` and `DAG_Y_Index` after each DAG use
New_MIN_Word_On_YWrap	0: Ignore `DAG_Y_Index` wrap when unpacking/packing MIN words
High_Speed_Write	1: High-speed mode; The port does not support read requests
Burst	1: High-throughput mode for accessing contiguous blocks of 2-D data
Random_Access	0: Normal DAG addressing when performing read request

Table IX Parameters for Burst-Write Mode

Where:

1. `ACK_Count` ≥ 0, indicating that there are words in the temporary buffer, triggers the processing of those words.

2. Once processing begins, the entire contents of the temporary buffer – as indicated by `ACK_Count` when processing begins – is processed.

3. Data words from the temporary buffer are unpacked from right to left[†] and the records written to main-memory under DAG direction. Upon an X Wrap (`DAG_X_Index`

wraps around), writing is immediately terminated and any remaining records in the data (double-) word are discarded.

4. There is no flow control between the temporary buffer and main memory and so new data may overwrite old.

5. When a control word is encountered in the temporary buffer, the indicated update is performed.

6. Upon completion of processing:

   a) ACK_Count is decremented by 4 x the total number of words – both data and control – removed from the temporary buffer

   b) Consumer_Count is incremented by 4 x the total number of data words written to main memory

   c) A **Backward Acknowledgement** is sent to Data Producer with a value equal to minus 4 x the total number of *data* words – if any – written to main memory

   d) A **Backward Acknowledgement** is sent to Control Producer with a value equal to minus 4 x the total number of *control* words – if any – that are processed

7. In the restriction above – *The Data Producer ACKs in multiples of 4 x ceil(DAG_X_Limit/4)* – ceil(DAG_X_Limit/4) is the number of double-words needed for each line in a block of pixels. 4 x ceil(DAG_X_Limit/4) is that number converted to bytes. The restriction guarantees that ACK_Count will always reflect an integral number of lines in the temporary buffer and the port will therefore always write an integral number of lines to memory.

8. Write Example: Suppose Record_Size = *byte*, DAG_Address_Mode = 2-D and the DAG is configured to address a 9x9 block of records. When the 3rd, 6th, 9th, 12th, 15th, 18th, 21st, 24th or 27th double-word of an incoming block is encountered, only the right-most byte – which is the 9th byte of a line – is written to memory because DAG_X_Index wraps immediately after that byte is written. The remaining three bytes in the double-word are discarded and writing of the next line in the block begins with a new double-word from the MIN. Notice that this incoming 9x9 block of pixels requires 27 double-words in Burst Mode, but only 21 double-words in Y-Wrap Mode.

APPLICATIONS

[123] The features of the XMC can be used to advantage in different ways depending on a specific application. For example, in a "data-sinking" application it is sometimes necessary to store information about system performance (e.g., statistics or an error log) in memory. The data may have to be stored in real time and prevented from being overwritten by subsequent data. An XMC port configured in Finite-Sink Mode can provide that capability. The parameter settings for this mode are shown in Table X, below.

[124] Real-time data are written into a buffer in memory until the buffer becomes full whereupon writing ceases. The data can be read at any time via a read request.

	Read/Write Port i
PTP/DMA_Mode	Finite-Sink Mode
Record_Size	double-word (32 bits)
Read_Count	read-block size (records)
Addressing_Mode	1-D
DAG_Origin	start of buffer
DAG_X_Index	read pointer (initialized to 0)
DAG_X_Stride	4 (bytes)
DAG_X_Limit	buffer size (bytes)
DAG_Y_Index	write pointer (initialized to 0)
DAG_Y_Stride	4 (bytes)
DAG_Y_Limit	buffer size (bytes)

Table X Data-Sinking Application

[125] Another application is known as "data sourcing". Applications sometimes require a large or unbounded stream of fixed data – pseudo-random data or a wave table, for example – during real-time operation.

[126]   To provide the stream an XMC port can be configured in Auto-Source Mode accessing a circular buffer in memory containing the fixed data configured according to Table XI. The fixed data – which is typically written into the buffer at system initialization –can be supplied automatically to the consumer node, the flow being governed by normal PTP flow control using Forwards and Backwards ACKs. Because the buffer is circular and Buffer_Read is turned off, the port provides an infinite source of data.

Read/Write Port i

PTP/DMA_Mode	Auto-Source Mode
Record_Size	double-word (32 bits)
Read_Count	read-block size (records)
Addressing_Mode	1-D
DAG_Origin	start of buffer
DAG_X_Index	read pointer (initialized to 0)
DAG_X_Stride	4 (bytes)
DAG_X_Limit	buffer size (bytes)
DAG_Y_Index	write pointer (initialized to 0)
DAG_Y_Stride	4 (bytes)
DAG_Y_Limit	buffer size (bytes)

Table XI Data-Sourcing Application

[127]   Another type of application may require implementation of "delay lines." For example, digital audio broadcast, personal video recorders, modeling of acoustics, etc., types of applications can require a signal to be delayed by a number of samples. This requirement usually means that there will always be a certain

57

minimum number of samples in the delay line once the line reaches steady-state operation (once the number of samples in the delay line reaches a threshold).

[128] A delay line is implemented using a single port configured in Buffer Mode with Record_Size set to double-word as shown in Table XII. The circular buffer in main memory is accessed by DAG_X_Index for reading and DAG_Y_Index for writing. The initial value of Consumer_Count determines the length/size of the delay line: it is initialized to minus the size of the delay, converted to bytes.

[129] For example, to implement a delay line of 1,000,000 double-words, a buffer of at least 4,000,000 bytes is allocated in memory and Consumer_Count is initialized to –4,000,000 as illustrated in Table . Because of the initial value of Consumer_Count, no output appears until at least 1,000,000 double-words have been written into the buffer and Consumer_Count has been incremented by a cumulative value of at least +4,000,000 (by Forward ACKs from the Data Producer). After that threshold has been reached and Consumer_Count has been driven non-negative, an auto read occurs.

[130] In this example, the consumer node expects to get data from the delay line in blocks of 100 double-words, and so Read_Count is set to 100 (records). Upon an auto read, 100 double-words are removed from the buffer and sent to the Consumer (assuming Producer_Count < 0). Consumer_Count is then decremented by 400 (bytes). If the new value of Consumer_Count is still non-negative, then another auto read occurs and the cycle is repeated. If the new value of Consumer_Count is negative, then reading is inhibited until additional double-words are written into the buffer and Consumer_Count is again driven non-negative.

[131] In summary, once the number of samples in the delay line reaches at least 1,000,000 and Consumer_Count becomes non-negative, Consumer_Count never drops below –400 and the number of double-words in the delay line never drops below 999,900.

Read/Write Port i

PTP/DMA_Mode	Buffer Mode
Record_Size	double-word (32 bits)
Read_Count	100 (records)
Consumer_Count	–4,000,000 (initial value in bytes)
Addressing_Mode	1-D
DAG_Origin	start of buffer
DAG_X_Index	read pointer (initialized to 0)
DAG_X_Stride	4 (bytes)
DAG_X_Limit	≥ 4,000,000 (buffer size in bytes)
DAG_Y_Index	write pointer (initialized to 0)
DAG_Y_Stride	4 (bytes)
DAG_Y_Limit	≥ 4,000,000 (buffer size in bytes)

Table XII Delay-Line Application

[132] Another type of application may require "data reordering" in which the elements in a block of data need to be reordered. Table XIII illustrates an application – sometimes called a corner-turner or corner-bender – that interchanges the rows and columns of a two-dimensional block of data. The application example uses two XMC ports – Write Port i and Read Port j – both accessing the same two-dimensional buffer in memory.

[133] For example, bytes can be written four at a time to memory by rows (lines) using Port i, which has the DAG, configured in 1-D mode. (2-D mode could have been used, but 1-D is simpler and generates the same sequence of addresses.) When the Data Producer receives acknowledgement from the XMC that all data has been written to main memory, it signals the Consumer to begin reading.

The Consumer sends a backwards ACK to XMC Port j thereby driving Producer_Count negative and enabling a read.

[134] Bytes are read from memory by columns using Port j with the DAG in 2-D mode. But because reading is by columns and not rows, the usual roles of DAG_X_Index and DAG_Y_Index are reversed. DAG_X_Index now indexes successive bytes in a column, and DAG_Y_Index now indexes successive columns in the 2-D block. More precisely,

DAG_X_Index   =   R X line-length

DAG_Y_Index   =   C

[135] where R and C are the row and column, respectively, of a byte in the 2-D block. After each byte is read, DAG_X_Index is incremented by line-length thereby accessing the next byte in the column. After the last byte in the column is read, DAG_X_Index reaches L X line-length, where L is the number of lines (rows) in the 2-D block. But L X line-length = buffer-size = DAG_X_Limit and therefore DAG_X_Index wraps around to 0 and DAG_Y_Index is incremented by 1. The cycle is repeated for each column until DAG_Y_Index = line-length = DAG_Y_Limit, the indication that the entire block has been read. When the Consumer receives the entire block of data, it signals the Data Producer to begin writing once again.

PTP/DMA_Mode	Write Port i	Read Port j
	High-Speed-Write Mode	Basic Mode
Buffer_Read	0	0
Buffer_Write	0	0
Record_Size	double-word	byte
Read_Count	–	read-block size (records)
Addressing_Mode	1-D	2-D
DAG_Origin	start of buffer	start of buffer
DAG_X_Index	–	0 (initial value)
DAG_X_Stride	–	line length (bytes)
DAG_X_Limit	–	buffer size (bytes)
DAG_Y_Index	write pointer (initialized to 0)	0 (initial value)
DAG_Y_Stride	4 (bytes)	1 (byte)
DAG_Y_Limit	buffer size (bytes)	line length (bytes)

Table XIII Data-Reordering Application

[136] The XMC allows interlacing, or multiplexing, of multiple data streams into a single data stream. In Table XIV two streams arriving on XMC Ports i and j are combined in memory and then read from memory via XMC Port k.

[137] In a preferred embodiment interlacing of the two streams is accomplished by writing bytes arriving on Port i to even byte addresses in the main-memory buffer, and writing bytes arriving on Port j to odd byte addresses. (Note that when DAG_Y_Index for Port i wraps around it returns to 0, but when DAG_Y_Index for Port j wraps around it returns to 1.)

[138] Synchronizing of writing and reading is accomplished using a double-buffering scheme in which the two Data Producers write into one half of the main-memory buffer while the Consumer reads the other half. To make the scheme work,

61

each Data Producer signals the Consumer when it receives acknowledgement from the XMC that buffer-size/4 bytes have been written into the main-memory buffer. When the Consumer receives a signal from each Data Producer, it sends a backwards ACK to XMC Port k thereby driving Producer_Count negative and enabling a read of the interlaced data. When the Consumer receives buffer-size/2 bytes of interlaced data, it signals each Data Producer that they are permitted to write into the buffer half just read.

	Write Port i	Write Port j	Read Port k
PTP/DMA_Mode	Y-Wrap Mode	Y-Wrap Mode	Basic Mode
Record_Size	byte	byte	word (16 bits)
Read_Count	–	–	buffer size / 4 (records)
Addressing_Mode	1-D	1-D	1-D
DAG_Origin	start of buffer	start of buffer + 1	start of buffer
DAG_X_Index	–	–	0 (initial value)
DAG_X_Stride	–	–	2 (bytes)
DAG_X_Limit	–	–	buffer size (bytes)
DAG_Y_Index	0 (initial value)	0 (initial value)	–
DAG_Y_Stride	2 (bytes)	2 (bytes)	–
DAG_Y_Limit	buffer size (bytes)	buffer size (bytes)	–

Table XIV Data-Interlacing Application

[139] Data de-interlacing (de-multiplexing) is accomplished whereby instead of merging two data streams into one, one data stream is separated into two.

[140] Table XV illustrates an application that reverses the interlacing operation described in the preceding section. The input data stream arrives on XMC Port i and the two de-interlaced streams exit the XMC via Ports j and k. De-interlacing is accomplished by reading even bytes in the main-memory buffer using Port j and odd bytes using Port k. (Note that when DAG_X_Index for Port j wraps

62

around it returns to 0, but when DAG_X_Index for Port k wraps around it returns to 1.)

[141] Synchronizing of writing and reading is accomplished using a double-buffering scheme in which the Data Producer writes into one half of the main-memory buffer while the two Consumers read the other half. To make the scheme work, the Data Producer notifies the Consumers when it receives acknowledgement from the XMC that buffer-size/2 bytes have been written into the buffer. When the two Consumers receive the signal, they each send a backwards ACK to their XMC read port thereby driving Producer_Count negative and enabling a read of the de-interlaced data. When each Consumer receives buffer-size/4 bytes of data, it notifies the Data Producer that reading of the half buffer has been completed. The Data Producer waits until it receives notification from both Consumers before it begins writing into the just-vacated half buffer.

	Write Port i	Read Port j	Read Port k
PTP/DMA_Mode	Y-Wrap Mode	Basic Mode	Basic Mode
Record_Size	word (16 bits)	byte	byte
Read_Count	–	buffer size / 4 (records)	buffer size / 4 (records)
Addressing_Mode	1-D	1-D	1-D
DAG_Origin	start of buffer	start of buffer	start of buffer + 1
DAG_X_Index	–	0 (initial value)	0 (initial value)
DAG_X_Stride	–	2 (bytes)	2 (bytes)
DAG_X_Limit	–	buffer size (bytes)	buffer size (bytes)
DAG_Y_Index	0 (initial value)	–	–
DAG_Y_Stride	2 (bytes)	–	–
DAG_Y_Limit	buffer size (bytes)	–	–

Table XV Data De-Interlacing Application

[142] Many video compression algorithms (e.g.,MPEG) require reading numerous rectangular blocks of pixels (bytes) from a frame buffer. Table le XVI

illustrates an application in which data are written sequentially into a frame buffer via XMC Port i and in which rectangular blocks within the frame are read via XMC Port j.

[143] A Data Producer for Port i writes data into the frame buffer line-by-line via Port i, and when it receives acknowledgement from the XMC that the entire frame has been written to memory, it notifies the Control Producer for Port j.

[144] A Control Producer for Port j then sends a separate read request (see Section **Error! Reference source not found.**) to Port j for each block of pixels to be read, the parameter-update value in the request being used to update DAG_Origin. This newly updated value for DAG_Origin determines the location of the block to be read. The remaining DAG parameters determine the size of the block to be read. Table illustrates the parameter settings for a 9 X 9 block of pixels (bytes).

	Write Port i	Read Port j
PTP/DMA_Mode	High-Speed-Write Mode	Basic Mode or Burst Mode
Record_Size	double-word (32 bits)	byte
Read_Count	–	81 (records)
Addressing_Mode	1-D	2-D
DAG_Origin	start of buffer	updated via read request
DAG_X_Index	–	0 (initial value)
DAG_X_Stride	–	1 (byte)
DAG_X_Limit	–	9 (bytes)
DAG_Y_Index	0 (initial value)	0 (initial value)
DAG_Y_Stride	4 (bytes)	line length (bytes)
DAG_Y_Limit	buffer size (bytes)	9 x line length (bytes)

Table XVI Frame-Buffer Application

[145] The XMC provides a scheme employing indirect addressing. In indirect addressing data is accessed in two steps: (1) an address (pointer) is used to access a second address (pointer) and (2) this second address is used in turn to access user data. The XMC implements indirect addressing via two tables, Table A and

Table B, both residing in main memory as shown in Table XVII. Table A – which is accessed via XMC Port j – contains pointers into Table B. Table B – which is accessed via XMC Port k – contains user data.

[146] Port j is configured in Auto-Source Mode and the entries in Table 1 are read automatically, in order, and sent via PTP control words from XMC Port j to XMC Port k. (Note the Consumer_ID and Consumer_Port for Port j.) Normal PTP flow control between Port j and Port k guarantees that the input buffer on Port k never overflows.

[147] Each entry in Table A has a format where bit 31 (TableAEntry[31]) is set to 1, bits 30-28 (TableAEntry[30:28]) are set to 001 and bits 27-0 are used for the new DAG_X_Index value. TableAEntry[30:28] = 001 indicates that DAG_X_Index[k] is to be updated with the value in TableAEntry[27:0]. TableAEntry[31] = 1 indicates that the update is to be immediately followed by a read of Table B.

[148] Port k responds to read requests from Port j as it would from any other source. It updates the appropriate DAG parameter – DAG_X_Index in this case – and then sends Read_Count records to the consumer of user data. Normal PTP flow control between XMC Port k and the data consumer guarantees that the data-consumer's input buffer never overflows.

	Read Port j (Table A)	Read Port k (Table B)
PTP/DMA_Mode	Auto-Source Mode	Basic Mode
Consumer_ID	XMC (Consumer_ID[0] = 1 indicating a control word)	consumer of user data
Consumer_Port	K	consumer port
Record_Size	double-word (32 bits)	user defined
Record Format	{ 1, 001, Pointer[27:0] } (Record[31] = 1 indicates a read request) (Record[30:28] = 001 indicates X_Index update)	user defined
Read_Count	½ size of the input buffer of Port k	typically 1 (record)
Addressing_Mode	1-D	1-D
DAG_Origin	start of Table A	start of Table B
DAG_X_Index	0 (initial value)	updated via read request from Port j
DAG_X_Stride	4 (bytes)	–
DAG_X_Limit	size of Table 1 (bytes)	–
DAG_Y_Index	–	–
DAG_Y_Stride	–	–
DAG_Y_Limit	–	–

Table XVII - Indirect-Addressing Application

[149] Although the invention has been described with respect to specific embodiments thereof, these embodiments are merely illustrative, and not restrictive, of the invention. For example, although a PIN has been described as a data transfer mechanism other embodiments can use any type of network or interconnection scheme.

[150] Any suitable programming language can be used to implement the routines of the present invention including C, C++, Java, assembly language, etc. Different programming techniques can be employed such as procedural or object oriented. The routines can execute on a single processing device or multiple processors. Although the steps, operations or computations may be presented in a specific order, this order may be changed in different embodiments. In some

embodiments, multiple steps shown as sequential in this specification can be performed at the same time. The sequence of operations described herein can be interrupted, suspended, or otherwise controlled by another process, such as an operating system, kernel, etc. The routines can operate in an operating system environment or as stand-alone routines occupying all, or a substantial part, of the system processing.

[151] In the description herein, numerous specific details are provided, such as examples of components and/or methods, to provide a thorough understanding of embodiments of the present invention. One skilled in the relevant art will recognize, however, that an embodiment of the invention can be practiced without one or more of the specific details, or with other apparatus, systems, assemblies, methods, components, materials, parts, and/or the like. In other instances, well-known structures, materials, or operations are not specifically shown or described in detail to avoid obscuring aspects of embodiments of the present invention.

[152] A "computer-readable medium" for purposes of embodiments of the present invention may be any medium that can contain, store, communicate, propagate, or transport the program for use by or in connection with the instruction execution system, apparatus, system or device. The computer readable medium can be, by way of example only but not by limitation, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, system, device, propagation medium, or computer memory.

[153] A "processor" or "process" includes any human, hardware and/or software system, mechanism or component that processes data, signals or other information. A processor can include a system with a general-purpose central processing unit, multiple processing units, dedicated circuitry for achieving functionality, or other systems. Processing need not be limited to a geographic location, or have temporal limitations. For example, a processor can perform its functions in "real time," "offline," in a "batch mode," etc. Portions of processing can be performed at different times and at different locations, by different (or the same) processing systems.

[154] Reference throughout this specification to "one embodiment", "an embodiment", or "a specific embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention and not necessarily in all embodiments.

Thus, respective appearances of the phrases "in one embodiment", "in an embodiment", or "in a specific embodiment" in various places throughout this specification are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics of any specific embodiment of

5   the present invention may be combined in any suitable manner with one or more other embodiments. It is to be understood that other variations and modifications of the embodiments of the present invention described and illustrated herein are possible in light of the teachings herein and are to be considered as part of the spirit and scope of the present invention.

10   [155]   Embodiments of the invention may be implemented by using a programmed general purpose digital computer, by using application specific integrated circuits, programmable logic devices, field programmable gate arrays, optical, chemical, biological, quantum or nanoengineered systems, components and mechanisms may be used. In general, the functions of the present invention can be

15   achieved by any means as is known in the art. Distributed, or networked systems, components and circuits can be used. Communication, or transfer, of data may be wired, wireless, or by any other means.

[156]   It will also be appreciated that one or more of the elements depicted in the drawings/figures can also be implemented in a more separated or integrated

20   manner, or even removed or rendered as inoperable in certain cases, as is useful in accordance with a particular application. It is also within the spirit and scope of the present invention to implement a program or code that can be stored in a machine-readable medium to permit a computer to perform any of the methods described above.

25   [157]   Additionally, any signal arrows in the drawings/Figures should be considered only as exemplary, and not limiting, unless otherwise specifically noted. Furthermore, the term "or" as used herein is generally intended to mean "and/or" unless otherwise indicated. Combinations of components or steps will also be considered as being noted, where terminology is foreseen as rendering the ability to

30   separate or combine is unclear.

[158]   As used in the description herein and throughout the claims that follow, "a", "an", and "the" includes plural references unless the context clearly dictates otherwise.   Also, as used in the description herein and throughout the claims that

follow, the meaning of "in" includes "in" and "on" unless the context clearly dictates otherwise.

[159]   The foregoing description of illustrated embodiments of the present invention, including what is described in the Abstract, is not intended to be

5   exhaustive or to limit the invention to the precise forms disclosed herein. While specific embodiments of, and examples for, the invention are described herein for illustrative purposes only, various equivalent modifications are possible within the spirit and scope of the present invention, as those skilled in the relevant art will recognize and appreciate. As indicated, these modifications may be made to the

10   present invention in light of the foregoing description of illustrated embodiments of the present invention and are to be included within the spirit and scope of the present invention.

[160]   Thus, while the present invention has been described herein with reference to particular embodiments thereof, a latitude of modification, various

15   changes and substitutions are intended in the foregoing disclosures, and it will be appreciated that in some instances some features of embodiments of the invention will be employed without a corresponding use of other features without departing from the scope and spirit of the invention as set forth. Therefore, many modifications may be made to adapt a particular situation or material to the essential scope and

20   spirit of the present invention. It is intended that the invention not be limited to the particular terms used in following claims and/or to the particular embodiment disclosed as the best mode contemplated for carrying out this invention, but that the invention will include any and all embodiments and equivalents falling within the scope of the appended claims.

25